Süsteemprogrammeerimine keeles C

Loeng 13

Milles räägime mis juhtub lõimede ja fork käsu kooskasutamisel, kuidas tekitada lõimepõhiseid muutujaid ning signaalidest. Lisaks veidi sorteerimata kasulikku infot.

Eksamiajad

3. jaanuar - IT-140
10. jaanuar - II-102

10:00 – 14:00

(konsultatsioonid 17. detsembri ja 7. jaanuaril)

Eelmises osas (Programmi käsureavalikud)

Kokkuleppeliselt antakse parameetreid ühetähelistena, nii. et ees on miinusmärk

```
myprog -a -b
myprog -ab
myprog -ba

myprog2 -i infile -ooutfile
```

Eelmises osas (getopt())

getopt() tagastab valikud ükshaaval
ei sõltu järjekorrast ega valikute kokku-lahkukirjutamisest
loeb vajadusel ka valikute parameetrid

muudab argv sees olevate argumentide järjekorda, kuni kõik tundmatud on lõppu liikunud

Eelmises osas(getopt() kasutamine)

getopt() kutsutakse reeglina välja tsüklis

getopt() poolt tagastatud väärtust vaadeldakse switch lauses, mis määrab programmi hilisemat tööd määravad muutujad

tundmatud programmiargumendid loetakse sisse eraldi tsüklis pärast esimese lõpetamist

Eelmises osas (getopt_long())

Olemuselt identne getopt() funktsiooniga

Võtab vastu argumente ka nende pikal kujul
(myprog --midagi --midagi-muud=parameeter)

getopt.h failis

Eelmises osas (getopt_long())

Pikad valikud kirjeldatakse struktuuriga:
struct option { name, has_arg, flag, val }

kui flag pointer, salvestub val väärtus seal näidatud aadressile; kui on NULL, tagastab funktsioon väärtuse val

Mõlema kasutamine on lõppkokkuvõttes töökindlam, kui argumentidele ise midagi välja mõtlema hakata

Eelmises osas (Mäluhaldus)

Protsessil on virtuaalne aadressruum (0 – vägasuur)

Võib sisaldada hõivamata "auke"

Jaotub *page*ideks

Iga *page* on kas reaalses mälus või mõnes teiseses kohas

Page fault katse lugeda mälust, mis põhjustab andmete teisesest kohast reaalsesse mällu lugemise

Eelmises osas (Segmendid)

text segment: programmitekst e. käivitatav kood
data segment: programmi andmete segment, exec hõivab seda ette, saab ise juurde võtta
stack segment: programmi *stack*; kasvab vajadusel, ei kahane

Eelmises osas (Mälu saamine)

exec() - programmitekst ja globaalsed, staatilised muutujad

programselt – automaatsed muutujad

malloc() – dünaamiline mälu

mmap() - mälu suurejooneline hõivaja

Eelmises osas (mmap())

poogib faili virtuaalse aadressiruumi külge

loetakse sisse ainult reaalselt loetav osa (*page fault* tõttu)

Eelmises osas (mäluhalduse jälgimine)

Mäluhõivet saab jälgida mtrace() ja muntrace() funktsioonipaari abil

GNU spetsiifiline, mcheck.h

MALLOC_TRACE keskkonnamuutuja ütleb failinime, kuhu andmed salvestuvad

mtrace käsk analüüsib hõivamisi ja lekkeid

Tänases osas

- Lõimed ja fork
- Lõimepõhised muutujad
- Signaalidest uuesti
- Kasutajaterminalist
- Debugimisest: Stack trace

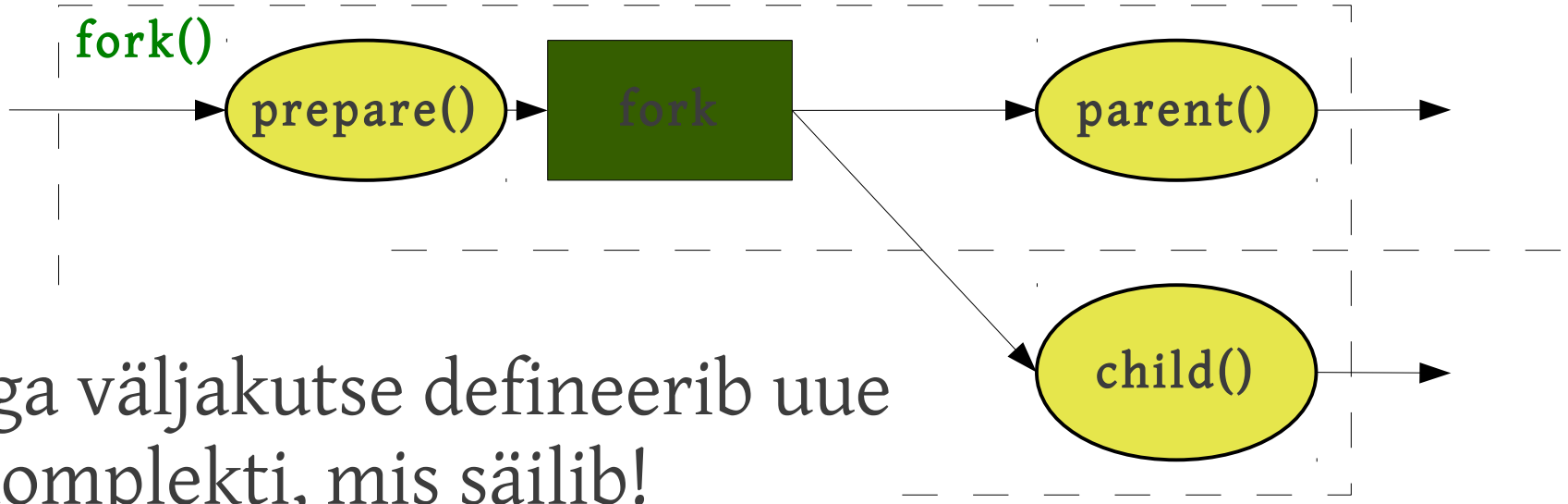
Lõimed ja fork()

- Lühikokkuvõte: paras jama tekib
- fork() kopeerib kogu protsessi mälu ja mõned süsteemsed objektid
- Ei kopeeri lõimesid
  - lapsprotsessis jookseb ainult fork() käsu 0-haru
- Ei hooli sünkroniseerimistest
  - võimalik, et kuskil on midagi poole kirjutamise peal
- Haldamiseks pakub POSIX funktsiooni pthread_atfork()

pthread_atfork()

- Registreerib käivituvad funktsioonid

```
int pthread_atfork (void (*prepare)(void),
                    void (*parent)(void),
                      void (*child)(void))
```



- Iga väljakutse defineerib uue komplekti, mis säilib!

pthread_atfork() (2)

- prepare – need funktsioonid käivituvad LIFO järjestuses
- parent, child – need funktsioonid käivituvad FIFO järjestuses
- Kuidas kasutada: prepare() lukustab mutexid ja parent() laseb nad uuesti lahti. child() initsialiseerib mutexid (ja muud sünkronisatsioonimuutujad) pthread_mutex_init() abil uuesti.
- Ükski koodijupp ei ole siis kriitilises regioonis, kui fork() käima läheb ja kopeeritakse korralik mälupilt.

Lõimed, Stream ja fork()

- Stream'ide kaitseks on standardlibras oma salamutex, mis fork() käsu puhul käima lastakse.
- Seda haldab fork() ise.
- Jagatud *stream*'e endid peab valvama programmeerija:
  - fork ajal ei tohi olla ükski stream teise protsessi käes:
    - flockfile() - paneb streami lukku
    - funlockfile() - laseb lukust vabaks
- Pärast fork() käsku vanem ja laps enam lukku ei jaga ja elu läheb „huvitavamaks“.

Lõimepõhised muutujad

- Kui on tarvilik, et protsessi eri lõimedel oleks eri väärtusega muutujad, tuleb selleks kasutada TSD andmeblokki mälus (*Thread Specific Data*).
- Sellesse pöördumine käib läbi *TSD key* nimeliste leiutiste.

```
int pthread_key_create(pthread_key_t *key,
                       void (*destr_function)(void));
int pthread_key_delete(pthread_key_t *key);
int pthread_key_setspecific(pthread_key_t key,
                            const void * pointer);
void * pthread_key_getspecific(pthread_key_t key);
```

Lõimepõhiste muutujate näide

```
/* Key for the thread-specific buffer */
static pthread_key_t buffer_key;

/* Once-only initialisation of the key */
static pthread_once_t buffer_key_once = PTHREAD_ONCE_INIT;

/* Allocate the thread-specific buffer */
void buffer_alloc(void) {
  pthread_once(&buffer_key_once, buffer_key_alloc);
  pthread_setspecific(buffer_key, malloc(100));  }

/* Return the thread-specific buffer */
char * get_buffer(void) {
  return (char *) pthread_getspecific(buffer_key); }

/* Allocate the key */
static void buffer_key_alloc() {
  pthread_key_create(&buffer_key, buffer_destroy); }

/* Free the thread-specific buffer */
static void buffer_destroy(void * buf) {
  free(buf); }
```

*Socket*ite paar

- Kui protsessil on tarvis suhelda iseendaga, tuleb appi socketpair():

```
int socketpair (int namespace, int style,
                int protocol, int filedes[2])
```

- namespace: AF_LOCAL
- style: SOCK_STREAM SOCK_DGRAM SOCK_RAW
- filedes: siia tuleb avatud suhtluspaari failideskriptorite massiiv
- Ühendus on erinevalt *pipe*'ist, kahesuunaline.
- Sobilik näiteks enne fork() käsku

Signaalid

- Mäluvärskendav selgitus: Signaalid on tarkvaralised katkestused programmile, mida iseloomustab signaalinumber.
- Signaalide tekkimine:
  - sünkroonselt: sinu programm saadab signaali iseendale
  - asünkroonselt: kuskilt väljastpoolt tuleb jama kaela
- Isedefineeritud signaal: SIGUSR1
- Isedefineeritud signaal2: SIGUSR2

Signaali elukäik

- Tekkimisel läheb signaal ootele (*pending*)
  - Valikud: Blokeeritud signaal jääb ootele, kuni protsess vastavat signaali uuesti vastu võtma hakkab. blokeerimata signaal saadetakse protsessile edasi
- Saabumisel vaadatakse mis teha
  - Valikud: Signaali haldama määratud funktsioon käima lasta; Ignoreerida (ignoreeritavat signaali ei saa blokeerida: visatakse kohe ära)
  - Kui ei ole öeldud, mida teha, on igal signaalil ettemääratud tegevuskava (ignoori või viska programm mälust välja)

Signaali haldamine

- Haldamisel kaks põhistrateegiat:
  - *Handler* teeb reeglina võimalikult vähe, näiteks määrab ühe globaalmuutuja. Programmi töö jätkub poolelijäänud kohast.
  - katkestada programmi töö ja minna tagasi mingisse ettemääratud kohta (longjump)
- Arvesta:
  - signaalid võivad tulla asünkroonselt
  - teine handler võib keset esimese jooksutamist käima minna (sama handler reeglina blokitakse)

Tagastuv *handler*

- Töö jätkub sealt, kus pooleli jäi
- Kui haldad viga, mis muidu programmi töö lõpetaks, on hea esimesel võimalusel väljuda, sest programm ei pruugi olla enam töövõimeline.
- Et säärane *handler* midagi teeks, peab ta muutma mõnd globaalset muutujat
  - andmetüüp: sig_atomic_t

Atomaarsus *handleris*

- Mälus olevate objektide manipuleerimine ei ole reeglina atomaarne tegevus. Kui keset katkestuse haldamist keegi vahele hüppab, läheb ralliks.
- sig_atomic_t on tüüp, mille lugemine-kirjutamine on katkestuste saabumise mõttes atomaarne
  - Tegelikkuses võid GNU C puhul eeldada atomaarseks: kõik, mis on int ja kõik, mis on vähem kui int. Samuti võid eeldada atomaarsust pointeritüüpide puhul.
- Maht sõltub süsteemist

Programmi lõpetav *handler*

- Veahalduse puhuks on kõige puhtam viis hallata vastav signaal ning siis toosama signaal raise() käsuga välja saata.

```
volatile sig_atomic_t fatal_error_in_progress = 0;

void fatal_error_signal(int sig) {
if(fatal_error_in_progress) raise (sig); /* nb:
blokitud parajasti */
fatal_error_in_progress = 1;

/* siin puhastustöö (terminalid tagasi, lapsed tappa,
lukustused maha)  */

/* handler maha ja viga uuesti */
signal(sig, SIG_DFL);
raise(sig);
}
```

Hüppega *handler*

- Probleemiks võib olla see, et kriitilised andmestruktuurid on poole kirjutamise peal
- Võimalikud lahendused:
  - blokeerida võimalikud signaalid kriitiliste andmestruktuuride kirjutamise ajaks
  - kriitilised andmestruktuurid pärast hüpet uuesti initsialiseerida (või korrigeerida)

Handler ja mitmekordne käivitamine

- Handleris tuleb olla väga ettevaatlik:
  - Kõik kasutatavad globaalmuutujad olgu defineeritud volatile võtmesõnaga.

    See välistab kompilaatoripoolse optimeerimise ja lugemisel loetakse tõesti õiget kohta, mis võib-olla on vahepeal muutunud.
  - Kui teed funktsioonivälja kutse, vaata et see funktsioon oleks mitmekordselt käivitatav.

    Mõni funktsioon hoiab kuskil oma sala-andmeid ja kui ta parajasti töötab, läheb kõik lörri.

Mitmekordne käivitatavus

- Funktsioon on mitmekordselt käivitatav (*reentrant*), kui ta kasutab andmeid, mis on ainult *stack*is.
- Funktsioon ei ole mitmekordselt käivitatav, kui:
  - kasutab globaalmuutujat, staatilist muutujat või dünaamiliselt hõivatud mäluosa, mida kuidagi üles otsib (Kasutada tohib, kui sinu programm kasutab funktsiooni *ainult* handleris, või kui blokid signaalid enne funktsiooni kasutamist)
  - kasutab mõnd etteantud objekti ja on oht, et muudetakse mõlemal korral sama (näiteks printf handleris)

Mitmekordne käivitatavus (2)

- Enamikes süsteemides ei ole malloc() ega free() taaskäivitatavad.
  - kas võtta handleri jaoks mälu varem valmis või free puhul teha lipuke, mis näitab põhiprogrammile, et mälu tuleb vabastada
- errno-d muutev funktsioon ei ole mitmekordselt käivitatav; vältimiseks pead handlerisse sisenedes errno salvestama ning pärast funktsiooni kasutamist taasväärtustama
- lugemine on ohutu, kui oled teadlik, et tegemist võib olla poolel kirjutamisel olevate andmetega

Mitmekordne käivitatavus (3)

- mälu kirjutamine on oma olemuselt ka ohutu, kui tead, et sa protsessi käigus midagi programmi jaoks ära ei käki

Signaali blokeerimine

- Kriitilistel hetkedel ei ole hea signaali saada
  - selleks blokeerime signaaliga ühised andmed enne nende kirjutamist/lugemist
  - blokeerime signaali ka signaali*handler*i poolt muudetavate andmete testimise eel
- Blokeerimist teostavad funktsioonid kasutavad andmetüüpi sigset_t (signal set).

```
sigemptyset(sigset_t *sigset);
sigfillset(sigset_t *sigset);
sigaddset(sigset_t *sigset, int signum);
sigdelset(sigset_t *sigset, int signum);
```

Signaali blokeerimine (2)

- Blokeerimine modifitseerib protsessi signaalimaski:

```
int sigprocmask (int how, const sigset_t *restrict set,
                 sigset_t *restrict oldset)
```

- Et asju mitte lihtsaks teha, on igal threadil oma signaalimask: pthread_sigmask()
- how: SIG_BLOCK, SIG_UNBLOCK, SIG_SET
- set: signaalimask, mida kasutame
- oldset: siia pointer eelmisele signaalimaski salvestuskohta, või NULL, kui too meid ei huvita

Näidis blokkimisest

```
volatile sig_atomic_t flag = 0; /* handler mudib seda siin */

int main (void) {
  sigset_t block_alarm;

  /* signaalmaski initsialiseerimine */
  sigemptyset (&block_alarm);
  sigaddset (&block_alarm, SIGALRM);

  while (1)
    {
      /* vaatame kas signaal tuli; kui tuli, tühistame lipu. */
      sigprocmask (SIG_BLOCK, &block_alarm, NULL);
      if (flag)
        {
          /* pole-veel-tulnud puhul tehtav möllamine */
          flag = 0;
        }
      sigprocmask (SIG_UNBLOCK, &block_alarm, NULL);

      ...
    }
}
```

Terminalist

- Kaks režiimi
  - Kanooniline (*canonic*) režiim
    - kasutaja saab oma rida redigeerida kuni saadab reavahetuse, pärast mida saadetakse tulemus programmile edasi (kernel puhverdab)
  - Mittekanooniline režiim
    - kõik terminalilt tulevad nupuvajutused saadetakse programmile edasi
- Terminalide omadused on paras teadus
  - termios.h sisaldab terminalimõjutusfunktsioone

Curses

- Curses on hulk funktsioone, mis võimaldavad terminalist sõltumatut lähenemist terminalile
- Defineerib akna mõiste.
  - Üks vaade võib sisaldada mitut akent
  - Töö käik: Initsialiseeri curses, tee aknad, mölla akendega, hävita aknad, pane curses kinni.
- curses.h või ncurses.h
- Eelmisel slaidil toodud paras teadus hõlbustub, kuid on ikkagi paras teadus

Debugimisest

- Üks küsimustest: kus programm kokku jooksis?
  - GDB debugger
    - kompileerida programm võtmega -g
    - gdb ./programminimi
    - > run (paneb käima)
    - > bt (backtrace)
  - Makro:

```
#define TRACE_MSG fprintf(stderr, "%s() [%s:%d]\n", \
  __FUNCTION__, __FILE__, __LINE__)
```

    - poetada seda makrot kuhu vaja; kui enam pole vaja, tühjaks

Backtrace jooksvast programmist

- Backtrace'i saab välja näidata suvalisel hetkel:

```
char ** backtrace_symbols (void *const *buffer, int size)
```

  - Funktsioon tagastab pointeri *stack*'i kirjeldavate stringide massiivile (mille mälu tuleb hiljem vabastada)
  - Kui kompileerida võtmega -rdynamic, on massiivis näha ka *stack*'is olevad funktsioonid
  - Ahvatlev SIGSEGV puhul (paraku on *handler*ist mõistliku *trace*'i saamine seiklus kerneli dokumenteerimata kohtadesse (mis on küll ära tehtud, kuid mitte eriti porditav))

Midagi suvalist